26장 ES6 함수의 추가기능

함수의 구분

ES6 이전에는 모든 함수를 일반 함수로 호출할 수 있을 뿐만 아니라 생성자 함수로도 호출할 수 있었습니다.

따라서 모든 함수는 constructor로 생성되며,

이로 인해 불필요한 프로토타입 객체도 함께 생성됩니다.

이러한 문제를 해결하기 위해 ES6에서는 함수의 사용 목적에 따라 세 가지 종류로 명확하게 구분하였습니다.

ES6 함수의 구분	constructor	prototype	super	arguments
일반함수	O	O	X	O
메서드	X	X	O	O
화살표 함수	X	X	X	X

메서드

ES6 이전 사양에는 메서드에 대한 명확한 정의가 존재하지않았다.

ES6사양에서는 메서드 축약표현으로 정의된 함수만을 의미한다.

특징

1. 인스턴스를 생성할수없는 non-constructor 이다.
2. prototype 프로퍼티가 없고 프로토타입도 생성하지않는다
3. 본연의 기능의 취지에 맞는 super을 사용할수 있다.

화살표 함수

기존의 function 키워드 대신 화살표를 사용하여 기존의 함수 정의 방식 보다 간략하게 함수를 정의할수 있다.

일반함수와 화살표 함수의 차이

1. 인스턴스를 생성할수없는 non-constructor 이다.

2. prototype 프로퍼티가 없고 프로토타입도 생성하지않는다
3. 중복된 매개변수 이름을 선언할 수 없다.
4. 화살표함수는 함수 자체의 this,arguments,super,new.target 바인딩을 갖지않는다.
    - 스코프체인상의 가장 가까운 상위함수중에서 화살표함수가 아닌 함수의 this,arguments,super,new.target 를 참조한다.

this

화살표함수의 this가 일반함수의 this와 다르게 동작하는 이유는

콜백함수 내부의 this 문제때문이다.

```
class Prefixer {
  constructor(prefix) {
    this.prefix = prefix;
  }

  add(arr) {
      //1
    return arr.map(function(item) {
      return this.prefix + item; //2
      // TypeError
    });
  }
}

const prefixer = new Prefixer('-webkit-');
console.log(prefixer.add(['transition', 'user-select']));
```

1. map 메서드에 전달된 콜백 함수 내에서 `this`는 `undefined`를 가리킵니다.
    - 클래스 내부의 모든 코드에는 strict mode가 암묵적으로 적용됩니다.
    - Strict mode에서 일반 함수로서 호출된 모든 함수의 내부에는 전역 객체가 아닌 `undefined`가 바인딩됩니다.
2. 따라서, 클래스 내부에서 map 메서드의 콜백 함수 내부에서도 `this`는 `undefined`가 됩니다.이는 코드 작성자가 원하는 `this`와 다를 수 있으므로, 코드가 의도한 대로

작동하지 않을 수 있습니다.

화살표 함수에서는 콜백 함수 내부의 `this` 문제를 해결하기 위해, 화살표 함수 내부에서 `this`를 참조하면 상위 스코프의 `this`를 참조하도록 설계되었습니다.

```
class Prefixer {
  constructor(prefix) {
    this.prefix = prefix;
  }

  add(arr) {
    return arr.map(item => this.prefix + item);
  }
}

const prefixer = new Prefixer('-webkit-');
console.log(prefixer.add(['transition', 'user-select']));
// [ '-webkit-transition', '-webkit-user-select' ]
```

이는 화살표 함수가 자신의 `this`를 가지지 않고, 대신 자신을 포함하는 외부 함수의 `this` 값을 그대로 사용한다는 것을 의미합니다.

이 특성 덕분에, 화살표 함수는 클래스 내부의 메소드나 콜백 함수 등에서 `this`가 예상한 대로 작동하도록 하는 데 유용하게 사용됩니다.

다만 해당 특성 때문에 메서드로 선언하게되면

```
const person = {
    name: 'min',
    sayHi: () => console.log(`Hi ${this.name}`)
}

person.sayHi();
//expect : Hi min
//result : Hi
```

메서드를 호출한 객체인 person을 가리키지 않고 상위 스코프인 전역의 this를 가리키기때문에 코드가 의도한 대로 작동하지 않을 수 있습니다.

고로 ES6 메서드 축약표현으로 정의한 ES6 메서드를 사용하는것이 바람직합니다.

```
const person = {
    name: 'min',
    sayHi () {
        console.log(`Hi ${this.name}`)
    }
}

person.sayHi();
//expect : Hi min
//result : Hi min
```

super

화살표함수는 함수 자체의 super 바인딩을 갖지 않습니다.

따라서 this와 마찬가지로 상위 스코프의 super를 참조합니다.

arguments

화살표함수는 함수 자체의 arguments 바인딩을 갖지 않습니다.

따라서 this와 마찬가지로 상위 스코프의 arguments 를 참조합니다.

REST 파라미터

REST 파라미터는 함수에 전달된 인수들의 목록을 배열로 전달 받습니다.

```
function foo(...rest) {
  console.log(rest);
}
```

```
foo(1, 2, 3, 4, 5);
// [ 1, 2, 3, 4, 5 ]
```

특징

1. 일반 매개변수와 Rest 파라미터는 함께 사용할 수 있습니다.

   이때 함수에 전달된 인수들은 매개변수와 Rest 파라미터에 순차적으로 할당됩니다.

```
function foo(param, ...rest) {
  console.log(param); // 1
  console.log(rest); // [2, 3, 4, 5]
}

foo(1, 2, 3, 4, 5);

function bar(param1, param2, ...rest) {
  console.log(param1); // 1
  console.log(param2); // 2
  console.log(rest); // [3, 4, 5]
}

bar(1, 2, 3, 4, 5);
```

2. Rest 이름 그대로 먼저 선언된 매개변수에 할당된 인수를 제외한 나머지 인수들로 구성된 배열이 할당된다.

   고로 Rest 파라미터는 반드시 마지막 파라미터여야 합니다.

```
function foo(...rest, param1, param2);

foo(1, 2, 3, 4, 5); // SyntaxError
```

3. Rest 파라미터는 함수 정의 시 선언한 매개변수 개수를 타나내는 함수 객체의 length 프로퍼티에 영향을 주지 않는다.

```
function foo(...rest) {};
console.log(foo.length); // 0

function bar(x, ...rest) {};
console.log(bar.length); // 1
```

4. ES6에서는 Rest 파라미터를 사용해서 가변 인자 함수의 인수 목록을 배열로 직접 전달 받을 수 있습니다

   고로 유사 배열 객체인 arguments 객체를 배열로 변환하는 번거로움을 피할 수 있다.

```
function sum(...args) {
  // Rest 파라미터 args에는 배열 [1, 2, 3, 4, 5]가 할당된다.
  // Array.prototype.reduce() 는 배열 메서드이기 때문에
  // 배열로 변환하지 않으면 사용할 수 없다.
  return args.reduce((pre, cur) => pre + cur, 0);

}
console.log(sum(1, 2, 3, 4, 5)); // 15
```

매개변수 기본값

ES6에서는 도입된 매개변수 기본값을 사용하여 인수체크 및 초기화를 간소화 할 수 있다.

```
function sum(x = 0, y = 0) {
  return x + y;
}

console.log(sum(1, 2)); // 3
console.log(sum(1));    // 1
```

기본값을 사용하여 값을 할당하지않아도 에러없이 사용가능합니다.